NASA/CR-97- 206032

Final Report

Development and Demonstration of an Ada Test Generation System

Kestrel Institute

August 8,1996
Contract NAS-9-19113

1. Summary

In this project we have built a prototype system that performs Feasible Path Analysis on Ada programs: given a description of a set of control flow paths through a procedure, and a predicate at a program point feasible path analysis determines if there is input data which causes execution to flow down some path in the collection reaching the point so that the predicate is true. Feasible path analysis can be applied to program testing, program slicing, array bounds checking, and other forms of anomaly checking.

FPA is central to most applications of program analysis. But, because this problem is formally unsolvable, syntactic-based approximations are used in its place. For example, in dead-code analysis the problem is to determine if there are any input values which cause execution to reach a specified program point. Instead an approximation to this problem is computed: determine whether there is a control flow path from the start of the program to the point. This syntactic approximation is efficiently computable and conservative: if there is no such path the program point is clearly unreachable, but if there is such a path, the analysis is inconclusive, and the code is assumed to be live.

Such conservative analysis too often yields unsatisfactory results because the approximation is too weak. As another example, consider data flow analysis. A du-pair is a pair of program points such that the first point is a definition of a variable and the second point a use and for which there exists a definition-free path from the definition to the use. The sharper, semantic definition of a du-pair requires that there be a feasible definition-free path from the definition to the use. A compiler using du-pairs for detecting dead variables may miss optimizations by not considering feasibility. Similarly, a program analyzer computing program slices to merge parallel versions may report conflicts where none exist.

In the context of software testing, feasibility analysis plays an important role in identifying testing requirements which are infeasible. This is especially true for data flow testing and modified condition/decision coverage.

Our system uses in an essential way symbolic analysis and theorem proving technology, and we believe this work represents one of the few successful uses of a theorem prover working in a completely automatic fashion to solve a problem of practical interest.

We believe this work anticipates an important trend away from purely syntactic-based methods for program analysis to semantic methods based on symbolic processing and inference technology. Other results demonstrating the practical use of automatic inference is being reported in hardware verification, although there are significant differences between the hardware work and ours. However, what is common and important is that general purpose theorem provers are being integrated with more special-purpose decision procedures to solve problems in analysis and verification.

We are pursuing commercial opportunities for this work, and will use and extend the work in other projects we are engaged in. Ultimately we would like to rework the system to analyze C, C++, or Java as a key step toward commercialization.

2. Technical Contributions

Our overall technical achievement was to demonstrate effective use of theorem proving technology for program analysis. Effective means that the prover executes with tolerable response times in an interactive system; that it discovers and exploits interesting program properties that could not be determined by the usual analysis methods, and that it scales to large problems.

Our detailed technical achievements include:

- New and effective methods for integrating symbolic evaluation and theorem proving techniques.
- New methods for integrating linear arithmetic decision procedures into a general theorem prover.
- New techniques for formalizing array operations within a theorem prover.
- Demonstration of the scalability of feasible path analysis on complex Ada programs.

We will not go into a description of the prototype system or its technical approach, but instead cite two papers included in the appendix. The first, called "Applications of Feasible Path Analysis to Program Testing" describes the overall capabilities, structure, and application of the system. The second describes our approach to combining symbolic analysis and theorem proving methods, which is one of the key technical advances of the work.

The references are:

Goldberg, and T.C. Wang, "Integration of Symbolic Evaluation and Specialized Inference Components for Software Analysis, 1995, submitted for publication.

Goldberg, A., Wang, T.C., Zimmerman, D.. Applications of Feasible Path Analysis to Program Testing, *Proceedings of the International Symposium on Software Testing and Analysis,* Seattle, WA, Aug. 1994.

3. Tasks and results

On this research effort we executed most task as initially planned. We spent considerably more effort than initially estimated on tuning the theorem prover to obtain as the best simplification results we could. As a consequence we did not perform two tasks: construction of a path optimizer and a test data generator. The table below summarizes our effort.

Task Designator	Task Description	Status
F.1.	ATG Advanced Prototype Development	
F.1.1.	Platform	done
F.1.2.	Ada Preprocessor	
F.1.2.1.	Parse and Static Analysis	done
F.1.2.2.	In-lining called sub-program units	done (task defn. modified, we skolemize rather than inline)
F.1.2.3.	"Slicing" with respect to control structure	not done, determined unnecessary
F.1.2.4.	Loop Analysis	loops unrolled, and abstracted
F.1.2.5.	Subset Checker (Stub)	done
F.1.2.6.	Axiom creation	done
F.1.2.7.	Control flow graph creation	done
F.1.3.	Path Regular Expression Generator	done w/ nice graphical interface that was not initially tasked
F.1.4.	Path Analyzer	
F.1.4.1.	Interface definition	done
F.1.4.2.	Control Flow Graph Processing	done
F.1.4.2.1.	Variable partitioning	done
F.1.4.2.2.	Variable substitution construction	done
F.1.4.2.3.	Data type axiomitization	done
F.1.4.3.	Back substitution procedure	done
F.1.4.3.1.	Path regular expression processing	done
F.1.4.3.2.	Control structures	all but forward goto, and execeptions
F.1.4.4.	Simplification	done, very substantial task
F.1.5.	Satisfiability Tester	done

F.1.6.	Path Optimizer	not done
F.1.7.	Test Data Generator	not done
F.2.	ATG Test Plan	done
F.3.	ATG Integration Plan	done
F.4.	Periodic Reviews and Final Report	done

4. Deliverables

We are including along with this report a combined User Manual and Installation Instructions. In order to run the prototype it must be loaded onto a SUN SPARC computer running under the Solaris Operating System. The system uses the Refine Ada language system. This is a commercial product requiring a license. The delivered system includes a Refine/Ada demonstration copy which can only be used for demonstration and evaluation purposes.